Mass Storage Flash for Code Storage Applications

AMD

White Paper

EXECUTIVE SUMMARY

AMD has ideal product solutions for applications requiring low, medium, or high density XIP (execute in place) random access NOR Flash or high density Mass Storage sequential access Flash. AMD NOR Flash products offer high speed random access with discrete control, address, and data pins to support code execution directly out of Flash. The AMD Mass Storage devices are less expensive than the NOR products, utilize separate control signals, have a multiplexed command, address and data bus, and support high speed sequential memory accesses. These features allow the Mass Storage devices to satisfy applications that require very high speed read and write performance at a lower cost than NOR based products, including code storage applications.

Applications

Non-volatile Flash memory devices are ideally suited for both code and data storage applications. If the system requires an XIP device, where the application software can execute directly out of the Flash, a NOR based Flash component is an excellent choice. In some applications considering an XIP non-volatile Flash, the access time of NOR Flash may be too slow for optimum system performance or the price is considered to be too high. For these applications shadow memory should be considered to complement (non-volatile) Mass Storage Flash, improving system performance while reducing system cost.

While many shadow memory techniques can get quite complicated, most designers utilize large amounts of inexpensive DRAM, therefore only the physical memory approach (page 2) is needed. For those astute designers striving for the ultimate in cost reduction, virtual memory approaches (page 3) should be considered.

Cost Benefits

Simpler manufacturing processes used to develop Mass Storage Flash allow for a lower cost per bit/device, than NOR based Flash solutions. AMD sells Mass Storage devices that are guaranteed to have no bad blocks which greatly simplifies the software infrastructure needed for shadow RAM applications. The additional cost of hardware/ firmware is offset by the lower cost of mostly good Flash. Different OPN (ordering part numbers) are used to designate whether or not the device contains known bad blocks. With the appropriate hardware and firmware in the system, any component with bad blocks can have the defective blocks “mapped out” of the Flash device by the system application.

Data Protection

A high degree of data protection can also be achieved if temporary code or data changes are made in a volatile memory resource instead of directly in Flash. If write operations are not permitted directly in the Flash, there is virtually no risk of a system virus or other system related anomaly corrupting the non-volatile information. In applications that have high security sensitivity, operating out of a volatile memory region helps to prevent unwanted changes to application code stored in the Flash until those changes can be verified as safe.

INTRODUCTION

This document discusses the application of code memory storage in non-volatile Flash in which the application code is shadowed to RAM for execution. There are a number of reasons that it may be desirable, or necessary, to transfer code out of non-volatile memory prior to normal system operation, like applications with self modifying code, high performance needs, or those that desire a lower system cost. These applications may involve very different types of non-volatile Flash memory technologies, but the system requirements and implementation for code storage, transfer, and execution are similar.

The process of transferring the application code from the non-volatile memory device to the volatile memory resource is typically referred to as memory “shadowing”. This is because the system application code stored in one memory area, is identically shadowed (or copied) into another memory region for execution. One additional benefit of shadowing memory from the Flash memory to another memory resource is that the code stored in the non-volatile device can be compressed to reduce the amount of non-volatile memory needed. Decompression of the stored information may be performed as the compressed code is transferred out of non-volatile storage.

Publication# **22292** Rev: **A** Amendment/**0**
Issue Date: **December 1998**

Self Modifying Code

For applications in which the application code is self modifying, the risk of Flash code corruption, makes it necessary to transfer some or all of the application code from the Flash device to non-volatile memory. In this case the Flash is capable of XIP and can store the initialization code needed to transfer the Flash code to the non-volatile resource.

High Performance Considerations

In the case of XIP capable Flash storing the application code, the executable code is transferred to shadow memory to provide higher performance operation. As an example, a high density 32 Mb Flash memory device may have a read access time of 70 ns operating in a system running at 33 MHz which requires zero wait state performance. Since the Flash memory device is too slow to allow zero wait state system performance, the application code stored in the Flash must be transferred into a separate memory resource with an access time or 25 ns or less, depending on the actual application. The executable code can be transferred to a high speed SRAM or SDRAM memory resource which is capable of faster read access. A boot loader program can handle transferring the application code from Flash to the high speed memory. Since the information only needs to be shadowed during system initialization, performance typically is not an issue.

Non-Volatile Memory Not Capable of XIP

When non-volatile memory is utilized which does not support XIP operations, there is no choice available but to transfer the application code to an off chip XIP memory resource for code execution. The non-XIP code storage may involve very low cost Mass Storage Flash, capable of sequential access only, or one of the Serial Flash memory technologies which can only transfer information as a serial bit stream. In both the Mass Storage and Serial Flash applications, the code that is stored in the device must be shadowed into an XIP capable memory resource before the system can execute the code. And, since the non-volatile memory device is incapable of performing any code execution, a small amount of non-volatile off chip memory capable of XIP to run the boot loader is needed. Or else a specialized ASIC that is capable of generating the proper signals to copy the contents of the Flash device to the off chip memory may be used.

PHYSICAL MEMORY APPROACH

Full Shadow Memory Technique

By far, the simplest shadow memory technique to employ at the system level is the full memory shadow technique. In this case the amount of shadow memory available must be large enough to hold all of the application software that was stored in the non-volatile memory area. As the system is initialized, all of the application code stored in the Flash device is transferred to the shadow memory with decompression being performed on the stored information as needed. In this case there is no real distinction between the two memory resources other than the fact that the volatile memory is XIP while the non-volatile memory is either non-XIP or lower performance memory. This application does not treat the non-volatile memory as a virtual memory resource, as described in the following section, but simply as a physical memory device that is only used during initialization. The full shadow memory technique provides the highest performance, but at the highest cost, of all the shadow memory applications discussed.

There may be no specialized hardware required in the system to implement this shadow memory procedure, and the memory transfer may be handled by software alone. The microprocessor, microcontroller, or ASIC simply copies one memory region to the other and then jumps into the shadow memory for code execution. Once all of the information in the Flash device is transferred to the shadow memory region, the non-volatile memory is no longer required and may be powered down or placed in a low power standby mode. Figure 1 shows how the system appears during initialization and normal operation.

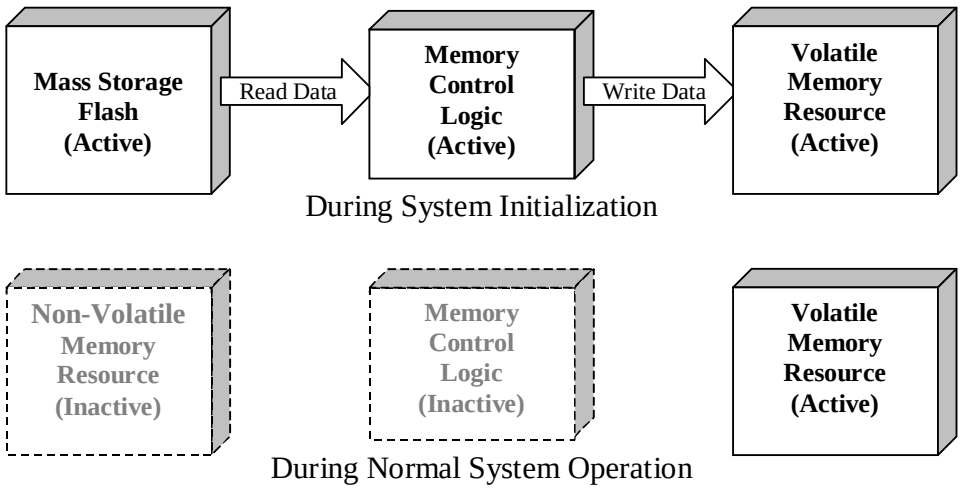

Figure 1. Shadow Memory Example

VIRTUAL MEMORY APPROACH

With Mass Storage non-XIP Flash approaching 128MB (megabyte) device densities, there is a strong incentive to utilize a smaller amount of volatile memory in the system to execute code stored in the Flash array. In these applications the volatile memory is treated as physical memory while the non-volatile Flash is treated as virtual memory. Where there is not enough off chip volatile memory to store all of the executable code, some form of paging algorithm must be employed. In these applications “demand paging” or “lazy evaluation” is typically used to move the necessary pages of code into the physical memory. Lazy evaluation simply indicates that a page of executable code is not moved into the volatile memory until it is actually needed for execution.

Virtual memory applications can be far more complex to implement than the full shadow memory technique. The system must determine when the necessary information is not available in the physical memory and needs to be paged in from the virtual memory resource. The system must also utilize an efficient method of determining which pages of memory are older or “stale,” so that frequently used code can remain resident in the physical memory. This allows infrequently used code to be replaced during paging operations. Once a virtual memory page is loaded into physical memory, address translation logic is needed so that virtual memory addresses can be converted to physical memory addresses. This address translation must be fully transparent to the system.

Demand Paging

Demand paging is somewhat analogous to a caching scheme used in most personal computer systems. In a memory read operation, if a memory access does not locate the information needed in the high speed SRAM cache, a cache line-fill operation is performed to move a small block of information from the lower performance main memory to the high speed cache memory. Since most code executes in a fairly linear sequence, and frequently loops into recently used code, the processor can usually find the information needed in the high speed cache. This is a classic example of demand paging where a page of information is not transferred until it is needed.

If a demand paging application is to be used, the system designer must select which algorithm will be used to determine which existing, valid, page of information is to be replaced with a newly requested page. There are a number of different algorithm classes to choose from and a few of the most common are included here. These are the DM (Direct Mapped), FIFO (First-In, First-Out), and LRU (Least Recently Used) algorithms.

Regardless of the algorithm used the system must employ some form of valid page logic to mark which virtual memory pages are resident in physical memory (valid) and which are not (invalid). The process of updating the pages in physical memory may be performed in hardware or software with hardware solutions being the fastest, or highest performance solutions. In a hardware application, when one of the page table valid bits indicates that the requested memory location is invalid, the memory cycle is put on hold and the appropriate page of information is copied from virtual memory to physical memory. In a software approach when the memory address falls within an invalid page, a page fault occurs which generates an interrupt to the system. A page fault interrupt handler would then execute which would move the appropriate page of information into physical memory. The interrupt handler would then transfer control back to the application to continue.

A simple form of valid logic may consist of a one bit wide SRAM with a separate addressable location for every page in the Flash device. When the system is initialized all of the SRAM locations are cleared to zero indicating that all pages in physical memory are invalid. Whenever a page is copied from virtual memory to physical memory the appropriate Valid Page Bit in the SRAM is set to one. Whenever a memory access is performed the valid bit is checked. If the valid bit is set to one the memory cycle is allowed to complete since the target page is valid in physical memory. If the valid bit is zero, the current memory cycle is suspended until the requested page is copied from virtual memory to physical memory, and then the suspended memory cycle is allowed to complete. This type of valid page logic is necessary whenever a paging application is used and would typically be handled by the system control logic.

Paging Algorithms

Each of the paging algorithms discussed will assume an extremely simple virtual and physical memory implementation. For convenience sake, each page in the virtual memory device is assigned a page size of 512 bytes, and the physical memory resource is 25% the size of the virtual memory device or array. For example, an application using 128 MB of Flash as virtual memory would have 32 MB of physical memory.

A simplified block diagram of this simple system implementation is shown in Figure 2. The virtual memory pages run numerically from the least significant device page (page 00) to the most significant page (page 15). Code that executes sequentially through the device would step from one page to the next higher order page.

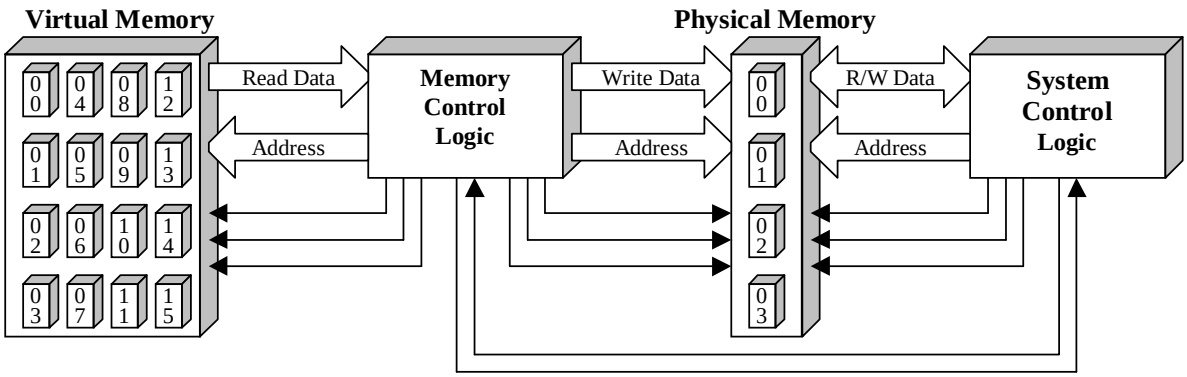

Figure 2. Virtual/Physical Memory Application (Example Block Diagram)

As the system executes the demand paging operations, the pages in physical memory will be replaced, or overwritten, with new information from virtual memory. How the system decides which page in physical memory is replaced depends on the paging algorithm chosen. In the simplified block diagram, the valid page logic is contained in the system control logic block. If the system control logic determines that the page is valid, the physical memory access completes normally. If the page is determined to be invalid, the memory cycle is put on hold and system control is passed to the memory control logic. The memory control logic would update the page in physical memory, mark the page as valid, and inform the system control logic that the cycle should be allowed to complete. The memory control logic determines which page in physical memory is to be replaced by a new page from virtual memory.

In system designs that involve paging algorithms, a trade-off is made between design complexity, physical memory resources required, and memory performance. The best memory performance is achieved with the fewest page faults incurred which cause a physical memory page update. The page replacement algorithm performance comparison chart shown in Figure 3 is intended to show the relative performance of the DM (least complex), FIFO (moderately complex), and LRU (most complex) page replacement algorithms. The performance of the paging algorithms are compared to the Full Shadow non-paging technique which provides zero page faults at the highest cost.

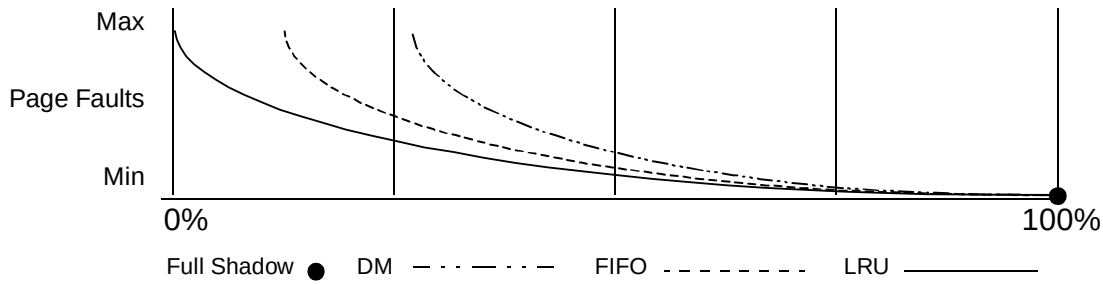

Figure 3. Page Replacement Algorithm Performance Comparison

DM—Direct Mapped

Of the various paging algorithms discussed here, the DM algorithm is the simplest to implement. The DM implementation allows only one specific destination page in the physical memory area for any one page in the virtual memory area. As shown in Figure 4, if any page in one of the virtual memory "rows" needs to be paged into the physical memory, it will displace another page in the same row that was previously loaded into physical memory. The DM application is not extremely efficient but usually provides reasonably high performance in most system applications.

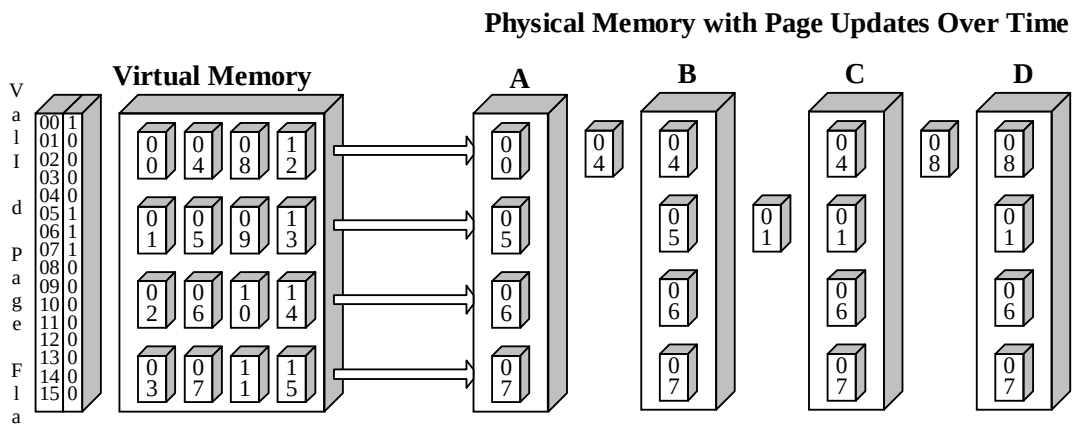

Figure 4. DM Page Replacement Algorithm

One of the benefits of this approach is that no logic is required to determine which page in physical memory should be discarded to load the new page from virtual memory. Since there is only one possible physical memory page location for each virtual memory page, the only page logic required is the valid page logic needed to determine if the target address in physical memory is valid or not.

The primary disadvantage to the direct mapped application is the fact that the physical memory may not be fully utilized. As an example, in an extreme case where the system boots up into page 00 and then only runs in pages 00 and 04, only the first page in physical memory will be used. Whenever the code moves from page 00 to 04 or back again, the system will need to wait while the memory controller moves the requested page from virtual memory to physical memory, thus wasting physical memory blocks 2, 3, and 4. Some analysis of the application software can be performed to better optimize the performance of the direct mapped paging scheme. By varying the percentage of physical memory to virtual memory densities, the system designer should be able to develop a system that does not leave too much of the physical memory unused, and also does not result in an excessive amount of page updates during code execution.

In order to provide a much higher memory system efficiency, it may be necessary to implement one of the more complex paging algorithms. The more advanced algorithms include additional logic to assure that the

entire physical memory resource is used. No valid page loaded into physical memory is discarded to make room for a new page until every page in physical memory is used. This requires that the additional logic determine which of the many pages available should be replaced by the new page.

FIFO—First In, First Out

The FIFO page replacement algorithm generally provides a higher physical memory page utilization and higher page hit rate than that provided by the direct mapped approach. In this application FIFO logic needs to be added to the memory controller logic to determine which page in physical memory should be overwritten with the information from the newly requested virtual memory page. As shown in Figure 5, any page in virtual memory that needs to be paged into physical memory, will displace the oldest page that was previously loaded into physical memory. This page replacement occurs only after all of the physical memory pages are used. The FIFO algorithm is not the most efficient algorithm available, but does provide better performance than the direct mapped page replacement algorithm.

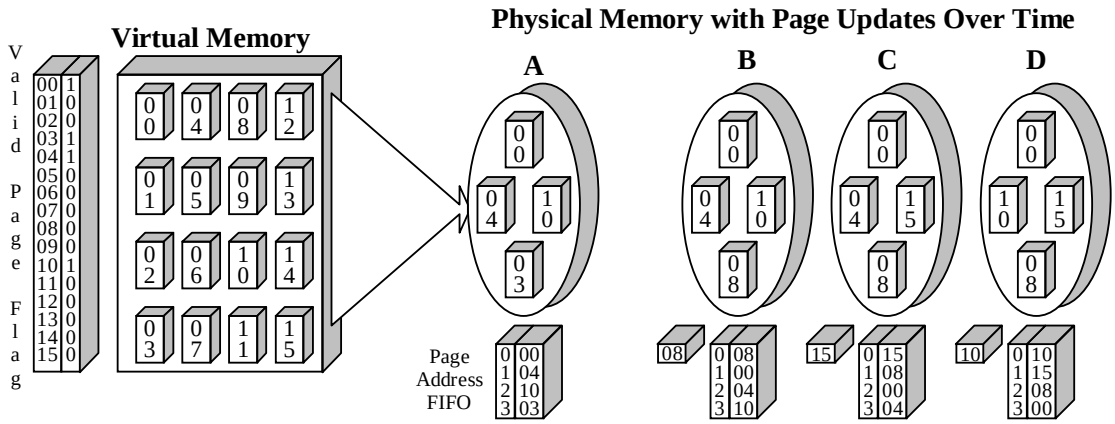

Figure 5. FIFO Page Replacement Algorithm

Some benefits of the FIFO approach are that all of the pages in physical memory are guaranteed to be used, any page in virtual memory can be stored in any of the physical memory pages, and the logic required to implement the FIFO is fairly basic. The FIFO implementation also does not restrict which locations the virtual memory pages can be written to in physical memory. Each time a new virtual memory page of information is loaded into physical memory, the address of the physical memory page loaded is clocked into a FIFO. This allows all of the physical memory pages to be filled before any page must be overwritten with a new page of information.

There are still limitations in the FIFO page replacement algorithm approach which can impact system performance. Unfortunately, the FIFO does not consider which pages in physical memory are used most often but simply replaces the oldest page when all pages are used and a new page needs to be loaded. If the first page loaded is where the application code spends a majority of its time, that page will still be replaced when a new virtual memory page is needed and physical memory is full. This can cause a number of pages that are accessed frequently to be bumped out of physical memory by pages of information that may only require a few bytes of information to be used.

Again, careful structuring of the application code can be performed to help alleviate this problem. Whenever possible the code should be structured so that most of a function, or software routine, can reside in contiguous memory pages. This will tend to limit the number of pages that need to be cycled into physical memory in order for the code to execute. The fewer pages that must be cycled into memory, the longer each page can reside in physical memory before being displaced.

The LRU or Least Recently Used page replacement algorithm can be used to improve demand paging performance at the hardware level above that provided by the FIFO algorithm and is discussed in the following section.

LRU—Least Recently Used

The LRU page replacement algorithm provides an improvement in memory system performance over the FIFO page replacement algorithm and is one of the most efficient algorithms available. The goal of the LRU page replacement algorithm is to determine which

of the pages in physical memory is the least recently used page. The assumption being that, based on past history, whichever page was least recently used should be the least probable page to need in the future. Since a future, or forward looking, algorithm (Optimal Algorithm) cannot be implemented the Least Recently Used algorithm is the best page replacement algorithm available.

The algorithm is typically implemented with a counter for each page in physical memory. Whenever a page is accessed in physical memory all other pages have their page counter incremented by 1. When it becomes necessary to replace a page in physical memory with a page from virtual memory, the physical memory page with the highest count is chosen for replacement.

The benefits of the LRU algorithm are that no pages in physical memory are replaced until the memory is full and only the least recently used page is chosen for replacement. This provides excellent system performance in virtually all applications.

The problem with the LRU page replacement algorithm is that it is generally prohibitively expensive to implement. The algorithm requires a separate counter for every page in physical memory. The counters must be updated once for every access to physical memory to a different page and are cleared when the associated page for that counter is accessed. An example of an LRU page replacement algorithm counter update scheme is shown in 1. There must also be a block of count comparison logic that is used to compare the contents of each page counter to determine which page has the highest count. This is the LRU page which should be replaced. There may be more than one LRU page counter with a maximum count since the counters are not of infinite length.

Table 1. Example LRU Page Replacement Scheme

Page Access	Page Counters				LRU
	Counter 0	Counter 1	Counter 2	Counter 3	Page(s)
Initialize	000	000	000	000	N/A
01	001	000	001	001	0, 2, 3
01	010	000	010	010	0, 2, 3
00	000	001	011	011	2, 3
02	001	010	000	100	3
03	010	011	001	000	1
02	011	100	000	001	1
03	100	101	001	000	1
03	101	110	010	000	1
02	110	111	000	001	1
02	111	111	000	010	0, 1
02	111	111	000	011	0, 1
01	111	000	001	100	0

For the simplified LRU implementation shown in Figure 6, there are four, three bit counters required for the physical memory array. In order to provide a better long term history of page accesses, the size of the counters can be increased accordingly. This example allows each page to indicate up to eight accesses since that page was last used.

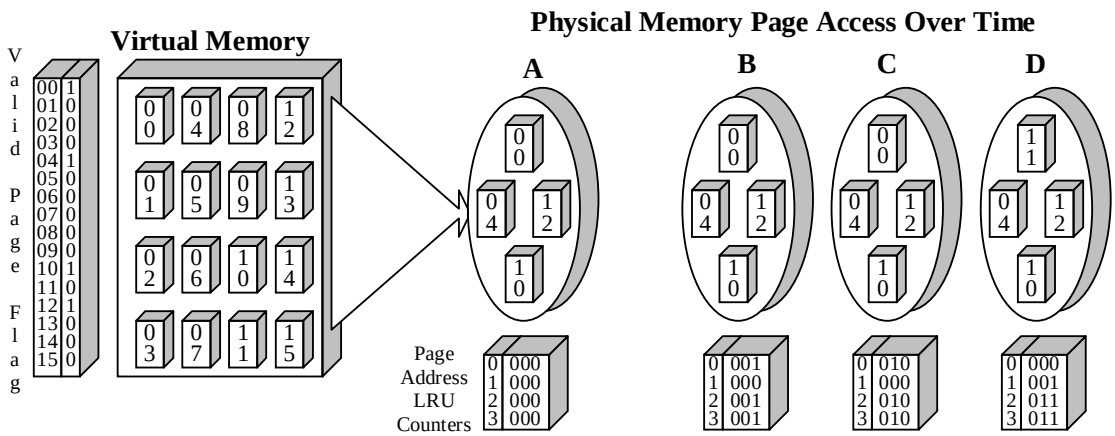

Figure 6. LRU Page Replacement Algorithm

Please note that in Figure 6 the page address LRU counters actually count physical memory page accesses and not page replacements or page updates. When the physical memory pages are all used, the LRU page replacement logic will assign one of the pages with the highest count to be replaced. When a physical memory page is either accessed or replaced the corresponding counter for that page is cleared to zero.

CONCLUSION

In high speed applications and those that require a lower overall system cost Mass Storage Flash can effectively be used as the system non-volatile code storage. Shadow memory methods like Full Memory, DM, FIFO, and LRU can provide the required system performance and achieve the system design goals.